An Assessment of a Beowulf System for a Wide Class of Analysis and Design Software

D. S. Katz, T. Cwik, B. H. Kwan, J. Z. Lou,
P. L. Springer, T. L. Sterling, and P. Wang

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

Abstract

This paper discusses Beowulf systems, focusing on Hyglac, the Beowulf system installed at the Jet Propulsion Laboratory. The purpose of the paper is to assess how a system of this type will perform while running a variety of scientific and engineering analysis and design software. The first part of the assessment contains a measurement of the communication performance of Hyglac, along with a discussion of factors which have the potential to limit system performance. The second part consists of performance measurements of six specific programs (analysis and design software), as well as discussion about these measurements. Finally, the measurements and discussion lead to the conclusion that Hyglac is suitable for running these types of codes (in a research/industrial environment such as at JPL,) and that the primary factor for determining how a given code will perform is that code's ratio of communication to computation.

Keywords: beowulf, pile of PCs, networked system, distributed system, software assessment, communications measurement, parallel performance, message passing.

1 Introduction

A typical Beowulf system, such as the machine at the Jet Propulsion Laboratory (JPL) may comprise 16 nodes interconnected by 100 Base-T Fast Ethernet. Each node may include a single Intel Pentium Pro 200 MHz microprocessor, 128 MBytes of I)RAM, 2.5 GBytes of IDE disk, and PCI bus backplane, and an assortment of other devices. At least one node will have a video card, monitor, keyboard, CD-ROM, floppy drive, and so forth. But the technology is evolving so fast and price performance and price feature curves are changing so fast that no two Beowulfs ever look exactly alike. Of course, this is also because the pieces are almost always acquired from a mix of vendors and distributors. The power of *de facto* standards for interoperability of subsystems has generated an open market that provides a wealth of choices for customizing one's own version of Beowulf, or just maximizing cost advantage as prices fluctuate among sources. Such a system will run the Linux[1] operating system freely available over the net or in low-cost and convenient CD-ROM distributions. In addition, publicly available parallel processing libraries such as MP1[2] and PVM[3] are used to harness the power of parallelism for large application programs. A Beowulf system such as described here, taking advantage of appropriate discounts, costs about $50K including all incidental components such as low cost packaging.

The Beowulf approach represents a new business model for acquiring computational capabilities. It complements rather than competes with the more conventional vendor-centric systems-supplier approach. Beowulf is not for everyone. Any site that would include a Beowulf cluster should have a systems administrator already involved in supporting the network of workstations and PCs that inhabit the workers' desks. Beowulf is a parallel computer, and as such, the site must be willing to run parallel programs, either developed in-house or acquired from others. Beowulf is a loosely coupled, distributed memory system, running message-passing parallel programs that do not assume a shared memory space across processors. Its long latencies require a favorable balance of computation to communication and code written to balance the workload across processing nodes. Within the constrained regime in which Beowulf is appropriate, it should provide the best performance to cost and often comparable performance per node to vendor offerings. This paper is intended to help determine where a wide variety of application codes fit within this regime.

To determine how a given message passing code will perform on a given machine, the communication characteristics of both the machine and the code must be known, as well as the computational performance of both the machine and the code. Computational performance of a code on a Pentiurn Pro system is under wide study, and will not be focused upon in this paper. Rather, characterization of the communications of the codes under study and the Beowulf machine at JPL will be the main theme of this paper, with computational performance being discussed as needed.

This paper considers many application codes. The codes span a wide spectrum of communication types, in terms of message size, message frequency, and message count. These codes were chosen to represent a sampling of the types of codes used at JPL. These include codes currently run on parallel supercomputers, sequential supercomputers, workstations, and PCs. The intent of this paper is to examine the performance of a Beowulf system on these codes, in order to determine the JPL-wide usefulness of this type of machine in helping engineers and scientists at JPL do their jobs quickly and well.

2 Beowulf Communication System

The Beowulf system at JPL comprises 16 nodes interconnected by a 16 port Bay Networks 281 15/ADV 100 Base-T Fast Ethernet switch. The network switch is built around a 1.6 Gbps switch fabric, thus allowing up to 8 simultaneous 100 Mbps streams between 8 pairs of nodes.

In this section, the communication attributes of the Beowulf system are examined. The metric used to measure performance is throughput. The communication attributes that affect throughput performance include the following:

- Packet size
- Traffic loading

Another attribute that affects overall performance is the ratio of communication to computation in a particular application. This ratio depends heavily on the application and details of this are left to the descriptions of the specific JPL application codes below.

2.1 Effects of Packet Size

An attribute of the network traffic that greatly influences throughput performance is packet size. In the parallel application codes described below, the a programming interface, Message Passing Interface (MPI), is used to support communication between nodes, Consequently, throughput performance is measured using the Linux implementation of MPI (MPICH from Argonne National Laboratory). In addition, because the MPI implementation of the communication calls is built on the Linux implementation of BSD sockets, throughput performance of BSD sockets is also measured to help evaluate the overhead associated with MPI.

To see the effects of packet size on an application using MPI, a simple test is run where one way communication latency is measured between two nodes using various packet sizes. Many of the communication calls for the JPL codes described below may be characterized by this type of interaction. The sending node uses a non-blocking send and is responsible for measuring the overall latency required to send 5000 packets of a specific size. The receiving node uses a blocking receive. The timer on the sending node begins immediately before the *MPI_Send* call is initiated and the timer stops immediately after the barrier call, which signifies that the receiving node has successfully received its packet. The overhead associated with the code surrounding the *MPI_Send* call (loop code, barrier call) is timed immediately before the actual test and is subtracted from the subsequent timings.

The implementation of the communication calls in MPI employ BSD sockets via the CH_P4 abstract device interface (ADI) used in the Linux implementation of MPI. The socket buffers are set to 64 Kbytes. Figure 1 shows the results of this experiment. As expected, the overhead of transmitting small packets degrades throughput. A steady rate of approximately 7 MBytes/s is only achieved when packet sizes rise above 8 Kbytes. A maximal throughput of 8.3 MBytes is achieved when the packet size is set to 256 Kbytes. A drop in throughput occurs when packet sizes are set to 32 Kbytes and 128 Kbytes but its cause is currently undetermined. This drop in performance may be linked to the socket buffer size and the ethernet segment size. The issue is still under investigation.

Also shown in Figure 1 is a similar experiment using BSD sockets. This experiment was run to show the overhead associated with MPI. In this study, net per f is used to measure the round trip latency of transmitting packets of varying sizes to determine the effective throughput. A TCP stream of packets is sent using netper f from a client node to a server node. The server node simply receives packets and immediately returns a packet to the sender. The client measures the round trip delay between transmission of the initial packet until the reception of the response packet from the server node. The performance of BSD sockets is clearly superior to that of MPI. A steady maximal rate of 11.8 MBytes is achieved for a packet size of 512 bytes and higher.

There are several reasons for the poorer MPI performance. One cause may be the overhead of setting up connections for each transmission under MP1. In the pure BSD socket implementation, a single connection setup call is executed and the set of packets are then transmitted. Other factors may include the segmentation processing that takes place at the socket level for packets larger than 64 Kbytes as well as at the OS device driver level to adapt to the use of ethernet which has a maximum transmission unit (MTU) of 1500 bytes.

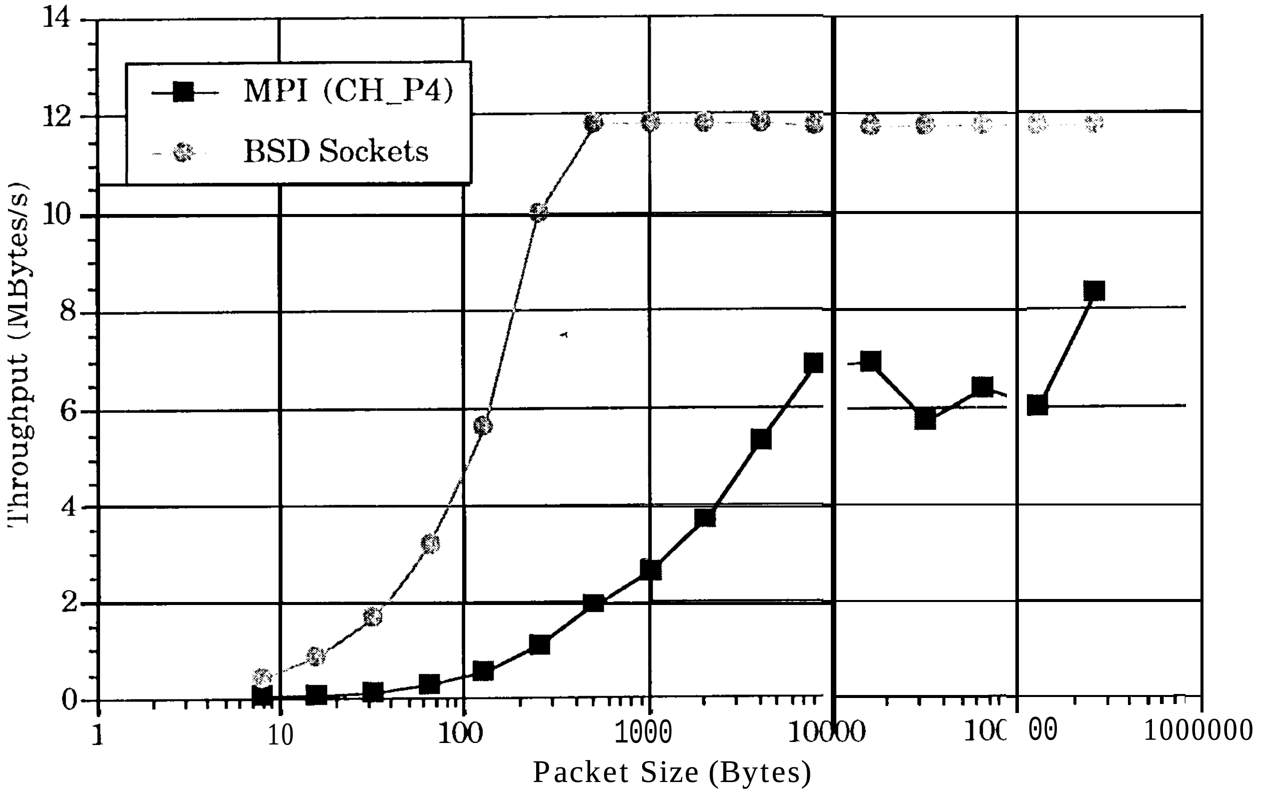


Figure 1. Packet Size effects on hroughput performance

2.2 Effects of Traffic Loading

‘I’here are various types of traffic loading that might affect performance of application codes by degrading throughput performance of the supporting network. In some systems, switching speeds and backbone network throughput is not sufficient to support multiple streams. Another type of traffic loading that may affect performance occurs when several streams attempt to send to the same node. Contention resolution in this case is handled both by the switch and the operating system that keeps track of the different connections flowing into the receiving node. Below, these two traffic loading situations are considered.

As mentioned above, the network switch is built around a 1.6 Gbps switch fabric and provides switching speeds that is able to keep up with 8 simultaneous streams with no degradation in performance. This means that designers of application codes do not need to be concerned about balancing the communications cost of supporting simultaneous connections.

The type of traffic loading where multiple streams are directed at a single node does affect throughput performance. Consequently, application codes that require a node to receive information from more than one other node do experience degradation in overall performance due to increased communication costs. This may occur when several nodes must report to a central node (all-to-one communication) or when a node receives packets from all of its neighbors (exchanging boundary conditions). Handling this contention is the responsibility of the switch and operating system which must efficiently multiplex the different streams to this node. In Figure 2, the results of an experiment are shown where

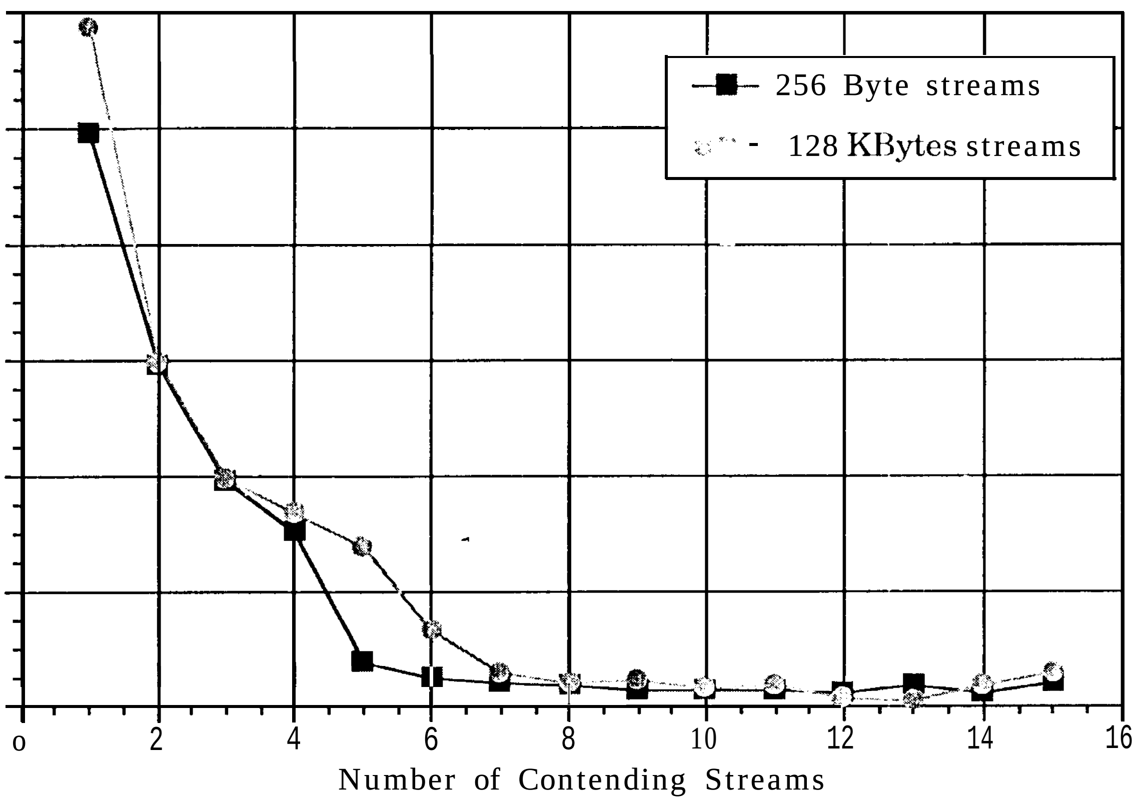


Figure 2. Traffic loading effects on throughput performance

contention is mapped against throughput for various packet sizes. In this experiment, netperf is used again to generate BSD socket streams from between 1 and 15 nodes to a single receiving node. Performance degrades rapidly and the results show that throughput drops to a minimum when there are more than 3 connections arriving at a single node. It should be noted that using a larger packet size would help improve performance slightly. This is due to the fact that when packets are large, a smaller fraction of time is spent multiplexing between streams as compared with delivering the data. In Figure 2, performance of larger packet sizes is shown to be primarily better when fewer than 7 streams are contending. Performance is approximately equal for the different packet sizes when the number of contending streams rises above 7.

3 Application Software

A *suite* of application software is considered in this assessment. It consists of a range of applications and related algorithms. There is also a range in the amount of data being communicated as well as the pattern of communication across processors. All applications use MPI or PVM for communication between processors and run on other platforms. 'l'he application codes are described by how they use communication and computation. Key parameters are: number of floating point operations, total number of operations, number of communication calls, frequency of communications calls, and length of communication calls.

3.1 Physical Optics Antenna and Telescope Design Software

The software described in this section[4] is used to design and analyze reflector antennas and telescope systems. It is based simply on a discrete approximation of the radiation integral[5]. This calculation replaces the actual reflector surface with a triangularly faceted representation so that the reflector resembles a geodesic dome. The Physical Optics (PO) current is assumed to be constant in magnitude and phase over each facet so the radiation integral is reduced to a simple summation. This program has proven to be surprisingly robust and useful for the analysis of arbitrary reflectors, particularly when the near-field is desired and the surface derivatives are not known.

Because of its simplicity, the algorithm has proven to be extremely easy to adapt to the parallel computing architecture of a modest number of large-grain computing elements such as are used in the Beowulf, or Intel Paragon parallel machines.

For generality, this code considers a dual-reflector calculation, which can be thought of as three sequential operations: (1) computing the currents on the first reflector using the standard PO approximation; (2) computing the currents on the second reflector by utilizing the currents on the first reflector as the field generator; and (3) computing the required field values by summing the fields from the currents on the second reflector. The most time-consuming part of the calculation is the computation of currents on the second reflector due to the currents on the first, since for N triangles on the first reflector, each of the M triangles on the second reflector require an N-element sum over the first. At this time, the code has been parallelized by distributing the M triangles on the second reflector, and having all processors compute identically the currents on all N triangles of the first reflector. Also, the calculations of field data have been parallelized. So, steps 2 and 3 listed above are currently performed in parallel, with step 1 being performed redundantly on each processor. Parallelization of step

1 will be performed in the future. There are also sequential operations in all three steps, such as 1/0 and triangulation of the reflector surfaces, some of which potentially could be performed in parallel.

Number of Processors	Beowulf			T3D		
	I	II	III	I	II	III
1	5.10	307	98.2	17.5	442	112
2	3.74	154	51.3	12.1	221	56.2
4	3.12	77.4	25.7	9.83	112	28.6
8	2.76	39.2	12.6	8.83	56.9	14.9
16	2.73	20.1	6.47	9.15	29.7	8.05

Table 1. Timing results (in seconds) for PO code, for M=40,000, N=400.

Number of Processors	Beowulf			T3D		
	I	II	III	I	II	III
1	0.0850	64.3	1.6	0.285	87.7	1.87
2	0.0624	32.2	0.838	0.202	44.0	0.937
4	0.0515	16.2	0.431	0.165	22.1	0.486
8	0.0459	8.17	0.211	0.148	11.2	0.243
16	0.0437	4.18	0.110	0.146	5.77	0.135

Table 2. Timing results (in minutes) for PO code, for M=40,000, N=4,900.

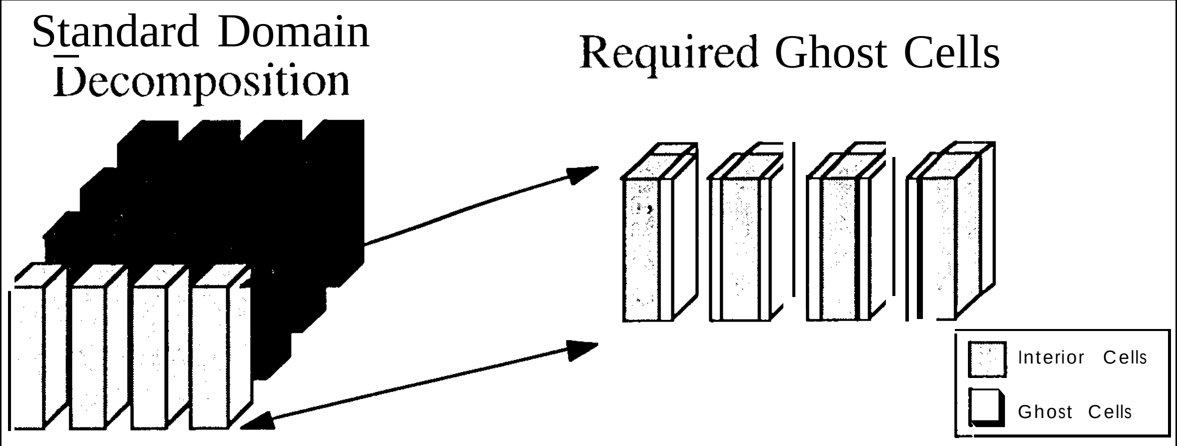
Tables 1 and 2 show timing results for the PO code, for 2 difference size sub-reflectors, with the same size main reflector. Each run is broken down into three parts. Part I is input 1/0 and triangulation of the main reflector surface, some of which is done in parallel. Part II is triangulation of the sub-reflector surface, evaluation of the currents on the sub-reflector, and evaluation of the currents on the main reflector. As stated previously, the triangulation of the sub-reflector and evaluation of the currents on those triangles is done redundantly, while evaluation of the currents on the main reflector is done in parallel. Part III is evaluation of far fields (parallel) and 1/0 (on only one processor).

It may be observed from Tables 1 and 2 that the Beowulf code performs better than the T3D code, both in terms of absolute performance as well as scaling from 1 to 16 processors, The absolute performance difference can be explained by the faster CPU on the Beowulf versus the T3D, and the very simple communication not enabling the T3D's faster network to influence the results. The scaling difference is more a function of 1/0, which is both more direct and simpler on the Beowulf, and thus faster. By reducing this part of the sequential time, scaling performance is improved. Another way to look at this is to compare the results in the two tables. Clearly, scaling is better in the larger test case, in which 1/0 is a smaller percentage of overall time.

As all the communication in this code is limited to a few global sums of very small length, it is clear that while the MPI packet size is small and the global sums are not performing at high communication throughput rates. Some overhead might also be expected since a global sum is an all-to-all communication, as discussed in section 2.2, but MPI global sums are performed intelligently, so that no node ever is receiving data from all (or even many) other nodes at one time. The overhead due to small message size is made unimportant by the extremely small ratio of communication to computation. Thus, this code performs very well, as expected, on Beowulf-class machines.

3.2 Finite-difference time-domain electromagnetic software

The software described in this section[6] is used for solving antenna patterns, calculating electromagnetic scattering from targets, or examining fields within small electronic circuits and boards. This version uses a uniform Cartesian grid, and describes the object being studied as a combination of cubic cells and square faces. The code uses an explicit time-marching scheme, and the parallelization is done by decomposing the 3-dimensional physical domain being studied over the processors, using a 2-dimensional decomposition, over x and y.



'igure 3. The relation between the 2-D decomposition of the 3-D grid and he required ghost cell communication.

This code has two types of communication. The first is interior communication, as shown in Figure 3. At each time step, fields at the boundary of each processor's sub-domain must be passed to neighboring processors for the next time step. This is the classic usage of ghost cells to hold data needed to update variables on one processor that themselves are updated on another processor.

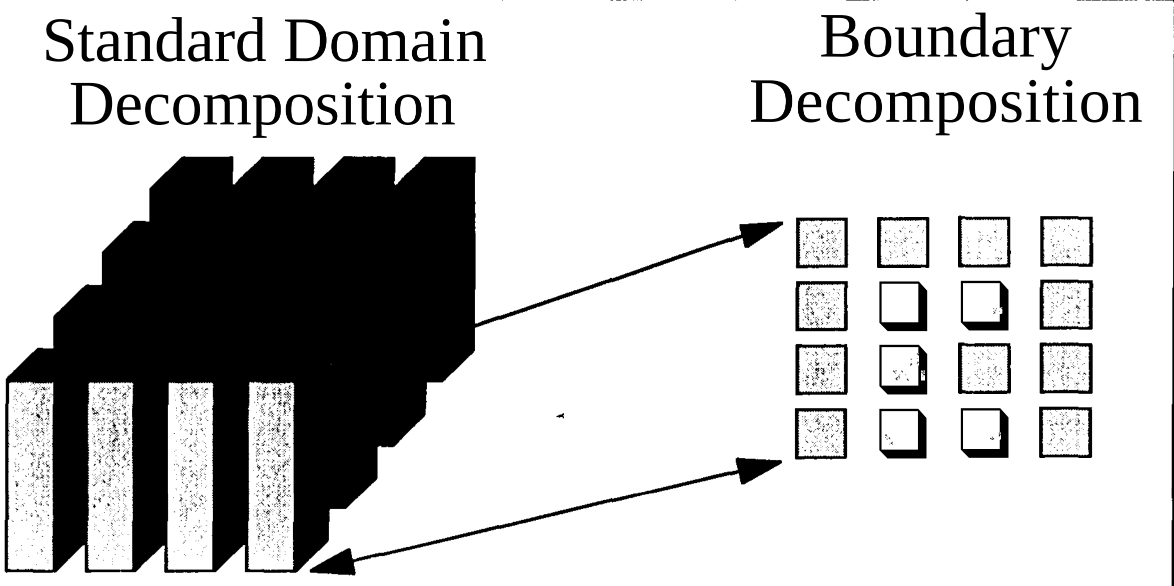


Figure 4. The relation between the 2-I) decomposition of the 3-D grid and the possible redistribution of the boundary variables.

The second type of communication is boundary communication, as shown in Figure 4. In this code, there is a large amount, of work that needs to be done at the exterior boundaries (enforcing boundary conditions, introducing a forcing function, and storing data used for post-processing output.) For the top and bottom boundaries, the variables to be worked on are distributed on all 16 processors. However, for the other four faces, the variables to be worked on reside on only 4 processors, and the other 12 processors will be idle while computations are being performed. Boundary communication solves this load balance problem, by redistributing the data on each of these four faces to all 16 processors, doing the required computations, and the returning the data to the original distribution.

Table 3 shows the relative amounts of communication and computation among the interior and boundary portions of the FDTD code. These values are for a problem with a global grid size of 282 x 362 x 102, which requires approximately half of the memory of each of the Beowulf processors, and is given in units of CPU seconds/time step. While this problem had a very good ratio of computation to communication on the Cray T3D, where it was developed, this ratio suffers on the Beowulf in that both the computation time is reduced and the communication time is increased.

During each time step, there are approximately 24,000 messages of 32 bytes, 90,000 messages of 200 bytes, and 12,000 messages of 800 bytes sent. The 32 and 200 byte messages are used in the boundary communication, while the 800 byte messages are used in the interior communication. All of these messages are relatively small, and it is reasonable that the communication time increases fairly dramatically. There are some potential problem with traffic loading at each node in an algorithm of this type, but the code is written in such a way as to sequentialize the communications in this situation to avoid this problem.

	T3D (shmem)	T3D (MPI)	Beowulf (MPI, Good Load Balance)	Beowulf (MPI, Poor Load Balance)
Interior Computation	1.8	1.8	1.1	1.1
Interior Communication	0.007	0.08	3.8	3.8
Boundary Computation	0.19	0.19	0.14	0.42
Boundary Communication	0.04	1.5	50	0.0
Total	2.0	3.5	55	5.5

Table 3. The amounts of communication and computation in the interior and boundary portions of the FDTD code, in CPU seconds per time step, for a 282 x 362 x 102 global grid size problem on 16 processors.

The last column of Table 3 was obtained by making the observation that using the boundary variables in place, and avoiding the multiple redistributions of this data, would not add significantly to the total amount of time spent in computation, and would have a dramatic effect on the amount of communication time. If the boundary computation time with perfect load balance is x, then with poor load balance (no data redistribution,) it should be $4(2x/3)+x/3$, or $3x$. This clearly is the better method for this code on the Beowulf machine, and indeed, would provide a significant improvement on the T3D (using MPI), as well. The code was not initially written using this method because the T3D shared memory message-passing library was used, since it provides substantially faster communication than MPI. Using this model, the redistribution of the boundary data made sense.

For a problem that more completely fills the memory of the Beowulf processors, the ratio of computation to communication increases. But even for this problem size, it is clear that modest rewriting of this code will result in Beowulf performance that is competitive with the T3D, at a substantially lower cost. (Note that while this FDTD code can be run on the T3D in 2.0 CPU seconds/time step, using a highly optimized kernel and the shared memory library communication routines, the cost difference still may make the Beowulf a good choice.)

3.3 Finite-element electromagnetic software

This software is used similarly to the finite-difference software, except it returns information at certain frequencies, rather than data over a time period. The complete finite-element code (PHOEBUS[7]) builds a large sparse matrix, distributes it over the processors, and solves for many right hand sides. After the building and distributing the matrix steps are completed on a workstation or one processor of a parallel computer, or multiple processors of a parallel computer using a data parallel language, the solution of the matrix for each of the right hand sides is done on the parallel computer using a code written within the message passing model, This section focuses on the matrix solve portion of the overall problem, since it usually requires over 98% of the complete problem time.

After the matrix is been built, reordered to minimize and equalize row bandwidth, and distributed into files, these files are read into the message passing matrix solution code. A block-like iterative scheme (quasi-minimal residual) is used for the matrix solve, in which a matrix-vector multiply is the dominant component. Figure 5 illustrates the process performed on a single processor in the matrix-vector multiply.

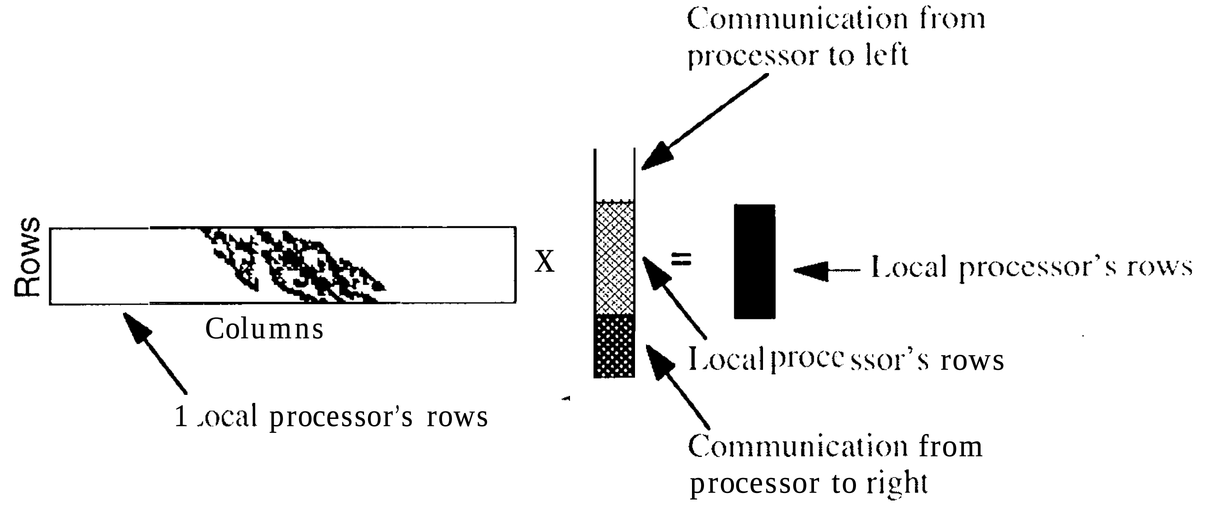


Figure 5. A representation of the work required in each processor to perform a matrix-vector multiply.

Because the matrix has been distributed in a one-dimensional processor grid, it makes sense to distribute the vectors similarly. The processor doing this portion of the multiply must acquire portions of the vector from its neighboring processors, as determined by the column extent of non-zeros in its portion of the matrix, and then perform the floating point operations of the multiply. The resultant vector is local to this processor, and no further communication is required.

Table 4 show performance data for this code, running on 16 processors, and solving a matrix problem formed from a model of a dielectric cylinder, with radius = 1 cm, height = 10 cm, permittivity = 4.0, and frequency = 5.0 GHz. The matrix is complex, with 43,791 rows having an average of 16 non-zero elements per row. The physical problem requires solving this matrix for 116 right hand sides. The items in this table are measurements of the time spent in each part of the matrix solution for all right hand sides, in CPU seconds.

	T3D (shmem)	T3D (MPI)	Beowulf (MPI)
Matrix-Vector Multiply Computation	1290	1290	590
Matrix-Vector Multiply Communication	114	272	3260
Other Work	407	415	1360
Total	1800	1980	5220

Table 4. CPU seconds required for each portion of the matrix solution for the test problem described in the text.

The computation in the matrix-vector multiply is 55% faster on the Beowulf CPU than on the T3D CPU. This is due to a combination of increased clock speed and increased cache size. The communication is about 10 times slower on the Beowulf compared with the T3D (both using MPI), which is to be expected for the size of the messages. This problem requires approximately 140,000 messages of 440,000 bytes, 78,000 messages of44,000 bytes, and 70,000 messages with length between 44,000 and 440,000, aswellas 40,000 global sums oflessthe160 bytes. With messages ofthis size, this code is obtainingas much throughput as is possible with MPI, and any potential problems with contending streams has been eliminated by the method in which the algorithm is implemented. All of the global sums are included in the data of the “Other Work” row, which also contains a large number of vector-vector operations (including dot products, norms, scales, copies.) The increase in time of this work from the T3D to the Beowulf can be viewed as a function of decreased memory-bandwidth, as there is almost no cache reuse in this work.

Overall, this problem is almost 3 times slower on the Beowulf than the T3D, due to a combination of communication speed, memory-bandwidth, and amount of communication. This is still decent price performance, but it is not outstanding, compared with the current value of a T3D.

3.4 **Incompressible fluid flow solver**

This software package implements a state-of-the-art numerical Navier-Stokes algorithm, a second-order projection method, for incompressible flow simulations on distributed-memory, message-passing machines. Features of this algorithm include its superior numerical stability in simulating high Reynolds number, non-smoothing flows and its robustness in resolving non-smooth, strongly sheared flows, due to the use of a Godunov scheme combined with an upwind scheme in the discretization of the convection term. The parallel flow solver package has been developed for solving two and three dimensional problems with a variety of boundary conditions on rectangular, staggered finite-difference grids. The model of our parallel implementation is domain partition and explicit message-passing. The parallel flow solver uses a parallel multigrid elliptic solver (also a stand-alone solver developed in-house at JPL) as a computation kernel to efficiently update velocity and pressure fields. A generic message-passing interface is used in the solver package with software wrappers implemented for several message-passing libraries, including MPI, PVM and Intel NX.

In the parallel implementation, grids at different levels (due to the use of multigrid schemes) are partitioned automatically in a preprocessing step based on the given physical (finest) grid. Due to the appearance of idle processors during processing on some coarse grids, a hierarchy of logical processor networks corresponding to the grid levels is created in the preprocessing step for message-passing at each grid level. The communication structure for the solver is constructed in a preprocessing routine and remains fixed during the solver execution. Flow simulations using the solver have been performed on the T3D to verify the numerical and flow physics results. Parallel performances in terms of speed-up and parallel scaling have been measured and studied on T3D and Intel Paragon systems[8].

A performance-comparison study using the solver *on* the Beowulf and T3D systems has also been performed, using the MPI version of the code. In the study, grids ofdifferent sizes are usedon 1,4, and 16 processors to compare the total execution time and parallel scaling. For this study, the flow solver is set up to run one time step in a 2-D driven-cavity flow in a unit square. A breakdown of computational cost and communication (message-passing) cost is also shown for each case. Table 5 shows the execution times on 16 processors on the Beowulf and the T3D for three grid sizes.

Grid Size	Number of Processors	Beowulf Run Time (seconds)	T3D Run Time (seconds)
64 x 64	16	12.1	3.6
256 X 256	16	22.7	9.6
1024x 1024	16	67.5	67.2

Table 5. Beowulf and T3D Results (Timing vs. Grid Size)

On the 64 x 64 grid, the T3D is about four times faster than the Beowulf. As the problem (grid) size increases, the Beowulf catches up with the T3D, and the execution times are almost the same for the 1024 x 1024 grid. The T3D wins on the small grid because its interprocessor communication bandwidth is higher than that of the Beowulf, and on the small grid, communication time dominates the total execution time. On the other hand the Beowulf wins on the large grid where local computations take a higher percentage of total execution time because the Pentium Pro processor of the Beowulf is faster in floating-point calculations than the Alpha processor on the T3D. Table 6 shows parallel scaling of the solver on the two systems, where each processor has a fixed grid size of 128 x 128 and the solver is run on varying numbers of processors.

Grid Size	Number of Processors	Beowulf Run Time (seconds): Total - computation - communication	T3D Run Time (seconds): Total - computation - communication
128x128	1	6.4- 6.4-0.0	13.8 -13.8-0.0
256 x 256	4	22.2 - 7.0 - 15.2	19.1 - 14.7 - 4.4
512 x 512	16	36.6 - 7.3 - 29.3	"""22; 7 - 15.4 - 7.3

Table 6. Beowulf and T3D Results (Timing vs. Number of Processors)

Perfect scaling would be indicated by constant execution time. The increase in execution time here is mainly due to the multigrid scheme used in the solver. When a larger global grid is used, there are more levels of grid to work on in the V-cycle and full V-cycles schemes, which increases both the computation and communication costs. Since the full V-cycle scheme (which is numerically more efficient than the V-cycle scheme) is used in the solver for these scaling tests, a larger global grid also implies more time is spent on coarse grids. This explains the increase in communication cost being more significant than that of computation cost. As can be seen, the T3D scales better than the Beowulf, which is again the result of faster communication bandwidth and slower processor speed on the T3D compared to the Beowulf. Running the code with a 1024 x 1024 grid on 16 processors produces 340,000 messages of size 24 to 48 bytes, 210,000 messages of 80 to 240 bytes, and 65,000 messages of 500 to 2000 bytes. There is some potential for traffic blocking at a single node, but since the communication is nearest-neighbor, at most four streams could be contending, and the application tried to prevent this from happening.

3.5 Non-linear thermal convection solver

This software is a parallel implementation of the finite volume method for three-dimensional, time-dependent, thermal convective flows. The discretization equations derived from the scheme, including a pressure equation which consumes most of computation time, are solved using a parallel multigrid method. In order to achieve load balance and to exploit parallelism as much as possible, a general and portable parallel structure based on domain decomposition techniques was designed for the three dimensional flow domain. It has l-D, 2-D and 3-D partition features which can be chosen according to different geometry requirements. MPI is used for communications. It currently runs on the Intel Paragon, the Cray T3D and T3E, the IBM SP2 and the Beowulf systems, and can be easily ported to other distributed memory systems.

The implementation of the software is based **on** the widely used finite volume method with an efficient and fast elliptic multigrid scheme for predicting incompressible fluid flows, which proved to be a remarkably successful implicit method. A normal staggered grid configuration is used and the conservation equations are integrated over a macro control volume. Local, flow-oriented, upwind interpolation functions have been used in the scheme to prevent the possibility of unrealistic oscillatory solutions at high Rayleigh numbers. A multigrid scheme is applied to the discretized equations, which acts as a convergence accelerator and reduces the CPU time significantly for the whole computation. The detailed parallel implementation and numerical results with Rayleigh numbers up to 10^7 have previously been published

Most of this code's communication occurs in each subdomain exchanging information with its neighbors (using message-passing) during each iterative level. Since only the values at the boundaries of each subdomain need to be updated at each iteration, the total amount of communication is relatively small as compared with the amount of computation (particularly when a large grid size is used.) The remainder of the communication occurs in the global sums which are used for the convergence and steady-state checks. These use much less time than the boundary communication. In this section, code performance is compared on the Cray T3D, the Intel Paragon, and the Beowulf. The focus is on the amount of computation and communication on these systems. Other performance data from the code, such as speed-up over different number of processors and comparison with other systems is available

	Paragon	Beowulf	T3D
Computation	110	35	27
Communication	49	60	19
Other Work	22	5	12
Total	181	100	58

Table 7. Run time in seconds for various portions of the solver for the test case described in the text.

Table 7 lists the detailed performance data of the amount of computation and communication on the three systems. Here a moderate grid size 64 x 64 x 64 with Rayleigh number equal to 10^6 in air was used for testing at fixed time steps by using 16 processors. The Cray T3D gives the best overall performance, and the Beowulf shows better performance than the Paragon. It is interesting to note the change of the ratio of computation to communication among these systems. It is obvious that the ratio changes rapidly from one system to another. On the Paragon, the code is dominated by the computation. On the Cray T3D most of the time is still used in computation, but the communication and other work (which includes the I/O and initial set-up) also use a fair portion of the total time. On the Beowulf, the code suffers by the slow network communication, and the communication becomes dominant. The computation speed is competitive with the Cray T3D for this problem, and, as the grid size of the problem is increased, the performance on the Beowulf will be improved. In conclusion, the Beowulf is a good system for computationally-intensive codes at a reasonable price.

3.6 **Parallel Extensions for Matlab**

Matlab[10] has become a popular tool among JPL scientists and engineers for matrix computation. It is a very capable tool, and offers a great deal of functionality and flexibility. However, certain problems are so large that they tax the computational resources of even the fastest workstations.

For these problems, the decision was made that parallel computation could provide a way to speed up the computation time. This section describes the software designed for this purpose, called Matpar. Matpar consists of Matlab extensions known as MEX-files, as well as code that runs on a parallel computer.

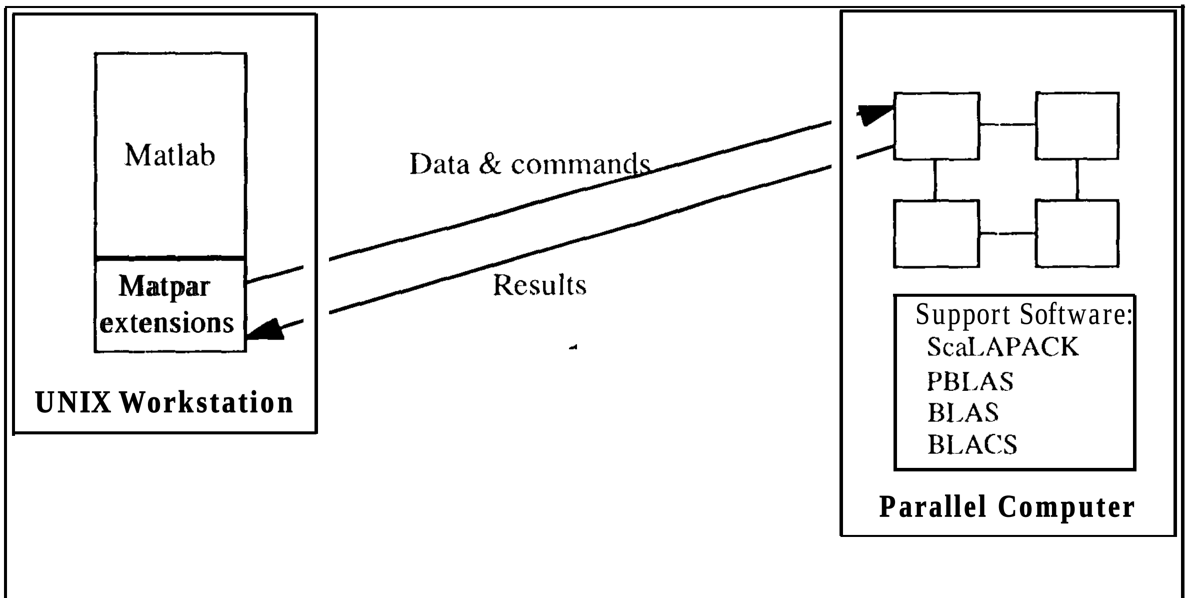


Figure 6. Matpar Architecture

The Matpar software follows a client/server approach. The client software resides on a workstation, and the server software is on a parallel computer (see Figure 6). The client software consists of a Matlab MEX-file for each of the parallel routines, as well as share'd object code. The MEX-files check the parameters appropriate for the call, and then call a corresponding routine in the shared code. The shared object code does some additional checking of the parameters, and then uses PVM to initiate a session on the parallel computer. Once the session has begun, the client transmits a Matpar request to the parallel computer, again using PVM communication routines. Each request contains the command to be executed, as well as all the necessary data for that command.

All communications between server and client go through a single node on the parallel computer, called the coordinator node. This decision was made because of the way PVM is implemented on one of the computers to which Matpar has been ported. In the Cray T3D version of PVM, a PVM connection is made only to the first node in the partition being used. In order to make Matpar as portable as possible, we decided to incorporate this characteristic into the software. This means that the data usually takes two hops, first from the client to the coordinator, and then from the coordinator to the final destination node.

The Matpar software is not a complete parallel version of Matlab. Instead it is targeted for very specific operations which require large amounts of computational power for large matrices. It is then left to the user to decide whether to call one of the parallel routines in place of an equivalent Matlab function. Because of the overhead involved in sending data from the workstation to the parallel computer, a user would not ordinarily call these routines for small matrices.

The speed obtained by using Matpar depends in large part on the ratio of computation to communication between client and server. The latter is done serially, and can take up more time than the computation for many problems. But where the ratio is high, very good results can be obtained. One Matpar benchmark problem runs thirty times faster using Matpar with 32 nodes of a T3D, than the same problem run using Matlab alone on a Sun UltraSparc.

In attempting to port the Matpar code to the Beowulf system, some problems have been encountered that have delayed the completion of the project. The first problem is that the individual nodes of the Beowulf do not have IP addresses that are visible outside of the cluster. In particular, they can not be addressed by the client, and so the PVM daemon running on the client can not add those nodes to its virtual machine. Vendors of massively parallel processors (M PPs) handle this situation by writing an MPP version of PVM that allows the MPP to be viewed as a single machine by PVM, but there is no such version of PVM written for the Beowulf computer.

A second problem with the Matpar port to Beowulf has to do with performance. The ScaLAPACK[11] routines take longer to execute on the Beowulf than on the Paragon. For example, a matrix-matrix multiply for a 1024 x 1024 double precision matrix distributed evenly across 4 nodes takes 32 seconds to execute on the Beowulf, compared with 13 seconds on the Paragon. Part of this performance problem can be explained by the difference in B LAS[12, 13, *4] libraries, which are highly optimized on the Paragon. The Paragon *dgemm* routine does a matrix multiply 28 times faster than a non-optimized C routine. The JPL Beowulf uses the public domain BLAS implementation, which runs only twice as fast as the non-optimized C routine. Pathcalc is a program used at JPL to estimate Beowulf run times based on varying values of CPU speed, network latency, anti bandwidth. Pathcalc estimates that if the Beowulf BLAS libraries were faster than non-optimized C code by the same factor of 28, one would see a run time of 6 seconds for the multiply operation.

The other part of the performance problem on the Beowulf relates to poor message passing performance. Matpar is dependent on PVM because of the spawning of jobs from the client to the server. Consequently, the PVM version of the BLACS message passing library (used by ScaLAPACK) is used, rather than the MPI version. Unfortunately, Matpar is seeing an effective bandwidth of only about 2 MB/s. Whether that is because PVM is slower than MPI, or it is due to the additional overhead of BLACS is not clear. If messages could be sent at MPI benchmark speeds, Pathcalc estimates the run time would drop to 29 seconds. Using both MPI speeds and better BLAS libraries would give a speed of 3 seconds, 4 1/2 times faster than the Paragon, and twice as fast as the T3D.

If the problem with the message passing speed can be resolved, the Beowulf platform has the potential to become one of the best platforms for Matpar because of its high computation speeds and large memory size (128 MB).

4 Conclusions

The intent of this paper was to discuss the Beowulf class of computers, focusing on communication performance, since the computation performance of the personal computer CPU which forms the building block of the Beowulf is well understood, and by describing a number of application codes and their performance on that class machine, to reach some conclusions about what this performance implies regarding the feasibility of using this type machine to run science and engineering codes in an institutional environment, such as JPL.

Potentially poor communication performance due to small message size, MPI overhead, and contention at a node have been explained, and six applications were introduced. It is observed that all the codes have been written in such a way as to remove the possibility of traffic contention, and that while communications performance is clearly a function of message size and thus important, the percentage of communication in each application has proven to be more important in estimating the overall performance of the applications.

The physical optics software had the best performance, primarily due to its almost embarrassingly parallel nature. The limited communications required by the software led to very good overall performance, due mostly to the CPU speed of the Beowulf being 33% faster than that of the T3D. This code is superior in both absolute performance and price-performance on the Beowulf than on the T3D.

The electromagnetic finite-difference time-domain software showed behavior that was found to be typical for most of the codes; that is, a reduction in communication performance and an improvement in computational performance. For this code, these did not balance out, but with some re-writing, a final code that performed within 60% of T3D performance was obtained. Given a cost difference of at least 1000% between the two machines, this is an extremely good price-performance comparison.

The electromagnetic finite-element software performed similarly to the finite-difference software, in that computation was faster, and communication was slower. The unique aspect of this code is the large amount of BLAS 1-type operations that are performed with data moved from main memory, rather than from cache. This work is substantially slower than similar work on the T3D. The reason for this is a combination of poorer memory-CPU throughput, and lack of optimized BLAS routines for the JPL Beowulf. Overall, this code has acceptable performance on the Beowulf.

The communication pattern in the flow solver is typical of a finite-difference PDE solver, much like the electromagnetic code, where the dominant communication is of the nearest-neighbor type. In the multigrid kernel, however, the communication is complicated by the appearance of idle processors on some very coarse grids. This irregularity of communication patterns is handled by setting up separate (logical) communication channels for grids/processors at different levels. The communication structure for the solver is constructed in a preprocessing routine and remains fixed during the solver execution. Overall, this code achieved reasonable performance on the Beowulf system, and in terms of price-performance, was clearly superior to the T3D.

The 3-D thermal convection code was successfully ported to the Beowulf system without any difficulty. The code is basically identical to the one running on the Paragon and the T3D. The performance on Beowulf is good if the code is numerically intensive and communication part is small. In spite of the slow communication rate associated with MP1 and the network used, the code's results as illustrated here demonstrate the great potential for applying this code to solving much higher Rayleigh number flow in realistic, three-dimensional geometries using the Beowulf system with a larger number of processors.

In its current configuration, Matpar has a good balance between computation and communication for its benchmark programs. However, both computation and communication times need to be improved for the overall execution speed to surpass the speed on the Paragon. This improvement can be made by using highly optimized BLAS libraries, and by either improving PVM performance on the Beowulf or using an MPI version of BLACS.

For MIMI) applications, such as Matpar, there is not a clear method for code assessment. It is possible that the ratio of communication to computation on the code's critical path (the path of operations which controls the complete execution time) may be the primary factor, but more codes of this type would have to be examined to make a final determination.

However, for most of the codes discussed in this paper, which more or less follow the SPMD model, the most important parameter in assessing each code's suitability to the Beowulf platform is the ratio of communication to computation. When this ratio is very small, the Beowulf has very good performance. As the ratio increases, the Beowulf still performs well when examined in terms of price-performance, but not in terms of absolute performance. For a communication-bound code, the Beowulf would be a poor choice.

Other communication characteristics such as packet size and traffic loading affect performance only minimally. Large packet sizes provide higher maximal throughput but may only benefit those codes that can be rewritten to use a few large messages rather then many small messages. Lastly, traffic loading may be an issue when the number of contending streams rises above a certain threshhold, but most codes and message-passing libraries are written specifically to avoid this problem.

5 References

1. *Red Hat Linux Unleashed,* Sams Publishing, Indianapolis, Ind., 1996.

2. Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W. & Dongarra, J., *MPI: The Complete Reference,* The MIT Press, Cambridge, Mass., 1996.

3. Giest, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., & Sunderam, V., *PVM: A Users' Guide and Tutorial for Networked and Parallel Computing,* The MIT Press, Cambridge, Mass., 1994.

4. Imbriale, W. A. & Cwik, T. A simple physical optics algorithm perfect for parallel computing architecture. In *10th Annual Review of Progress in Appl. Comp. Electromag., 1994,* Monterey, Cal., 434-441.

5. Imbriale, W. A. & Hodges, R. Linear phase approximation in the triangular facet near-field physical optics computer program. *Appl. Comp. Electromag. Sot. J., 1991,6.*

6. Woo, A. C. & Hill, K. C. The EMCC/ARPA massively parallel electromagnetic scattering project, NAS Tech. Report NAS-96-008. NASA Ames Research Center, 1996.

7. Cwik, T., Katz, D. S., Zuffada, C. & Jamnejad, V., The application of scalable distributed memory computers to the finite element modeling of electromagnetic scattering and radiation. *Int. J. Num. Meth. Eng.,* in press.

8. Lou, J. Z. & Ferraro, R. A parallel incompressible flow solver package with a parallel elliptic multigrid kernel. *J. Comp. Phy.,* 1996, 125, 225-243.

9. Wang, P. Massively parallel finite volume computation of three-dimensional thermal convective flows," to appear in the proceedings of *4th NASA National Symposium on Large-Scale Analysis and Design on High-Performance Computers and Workstations,* Williamsburg, Virg., Oct. 15-17,1997.

10. *MATLAB User's Guide,* The Mathworks, Inc., Natick, Mass., 1993.

11. Choi J., Dongarra, J. J., Pozo, R., & Walker, D. W. ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation,* IEEE Comp. Society Press, 1992, 120-127.

12. Lawson, C. L., Hanson, R. J., Kincaid, D. R. & Krogh, F. T. Basic linear algebra subprograms for FORTRAN usage, *ACM Trans. Math. Soft., 1985, 5,308-323.*

13. Dongarra, J. J., Du Croz, J., Hammarling, S. & Hanson, R. J. An extended set of FORTRAN basic linear algebra subprograms, *ACM Trans.* Math. Soft., 1988,14, 1-17.

14. Dongarra, J. J., Du Croz, J., Duff, I. S. & Hammarling, S.A set of level 3 basic linear algebra subprograms, *ACM Trans. Math. Soft., 1990,* **16,** 1-17.

6. Tables

Number of Processors	Beowulf			T3D		
	I	II	III	I	II	III
1	5.10	307	98.2	17.5	442	112
2	3.74	154	51.3	12.1	221	56.2
4	3.12	77.4	25.7	9.83	112	28.6
[illegible]	2.76	39.2	12.6	8.83	56.9	14.9
16	2.73	2(-.1	6.47	9.15	29.7	8.05

Table 1. Timing results (in seconds) for PO code, for M=40,000, N=400.

Number ot Processors	Beowult			T3D		
	I	II	111	I	II	111
1	0.0850	64.3	1.64	0.285:	87.7	1.87
[illegible]	0.0624	32.2	0.838	0.202	44 .0	0.937
[illegible]	0.05 1	16.2	9? !	0.165	22.1	0.486
8	0.0459!	8.17	0.211	0.148	11.2	0.2.43
16	0.0437	4. 18	0.11 0	0.146	5.77	0. 135

Table 2. Timing results (in minutes) for PO code, for M=40,000, N=4,900.

	T3D (shmem)	T3D (MPI)	Beowulf (MPI, Good Load Balance)	Beowulf (MPI, Poor Load Balance)
Interior Computation	1.8	1.8	1.1	1.1
Interior Communication	0.007	0.08	3.8	3.8
Boundary Computation	0.19	0.19	0.14	0.42
Boundary Communication	0.04	1.5	50	0.0
Total	2.0	3.5	55	5.5

Table 3. The amounts of communication and computation in the interior and boundary portions of the FDTD code, in CPU seconds per time step, for a 282 x 362 x 102 global grid size problem on 16 processors.

	T3D (shmem)	T3D (MPI)	Beowulf (MPI)
Matrix-Vector Multiply Computation	1290	1290	590
Matrix-Vector Multiply Communication	114	272	3260
Other Work	407	415	1360
Total	1800	1980	5220

Table 4. CPU seconds required for each portion of the matrix solution for the test problem described in the text.

Grid Size	Number of Processors	Beowulf Run Time (seconds)	T3D Run Time (seconds)
64x64	16	12.1	3.6
256x256	16	22.7	9.6
1024x1024	16	67.5	67.2

Table 5. Beowulf and T3D Results (Timing vs. Grid Size)

Grid Size	Number of Processors	Beowulf Run Time (seconds): Total - computation - communication	T3D Run Time (seconds): Total - computation - communication
128x 128	1	6.4- 6.4-0.0	13.8 -13.8-0.0
256 x 256	4	22.2 -7.0-15.2	19.1 -14.7-4.4
512 x 512	16	36.6 -7.3-29.3	22.7 -15.4-7.3

Table 6. Beowulf and T3D Results (Timing vs. Number of Processors)

	Paragon	Beowulf	T3D
Computation	110	35	27
Communication	49	60	19
Other Work	22	5	12
Total	181	100	58

Table 7. Run time in seconds for various portions of the solver for the test case described in the text.

7. Figures

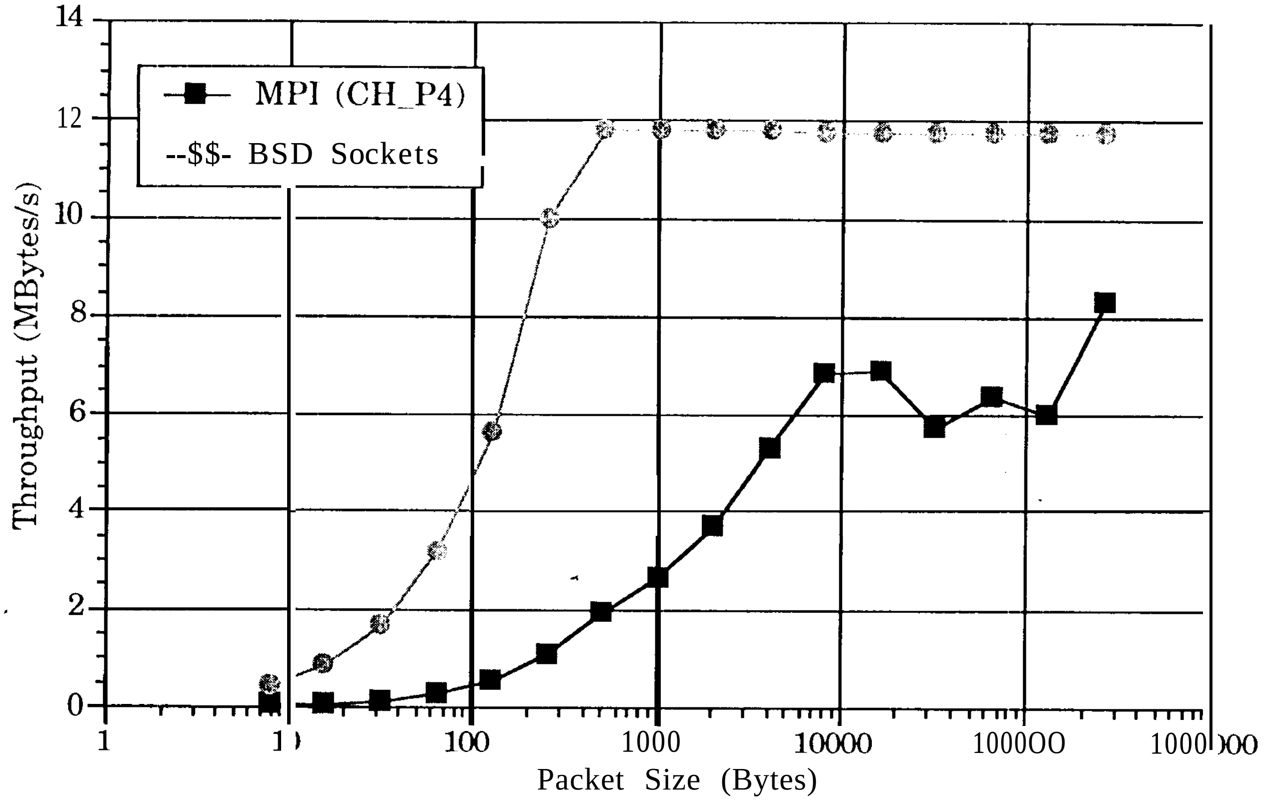


Figure 1. Packet Size effects on throughput performance

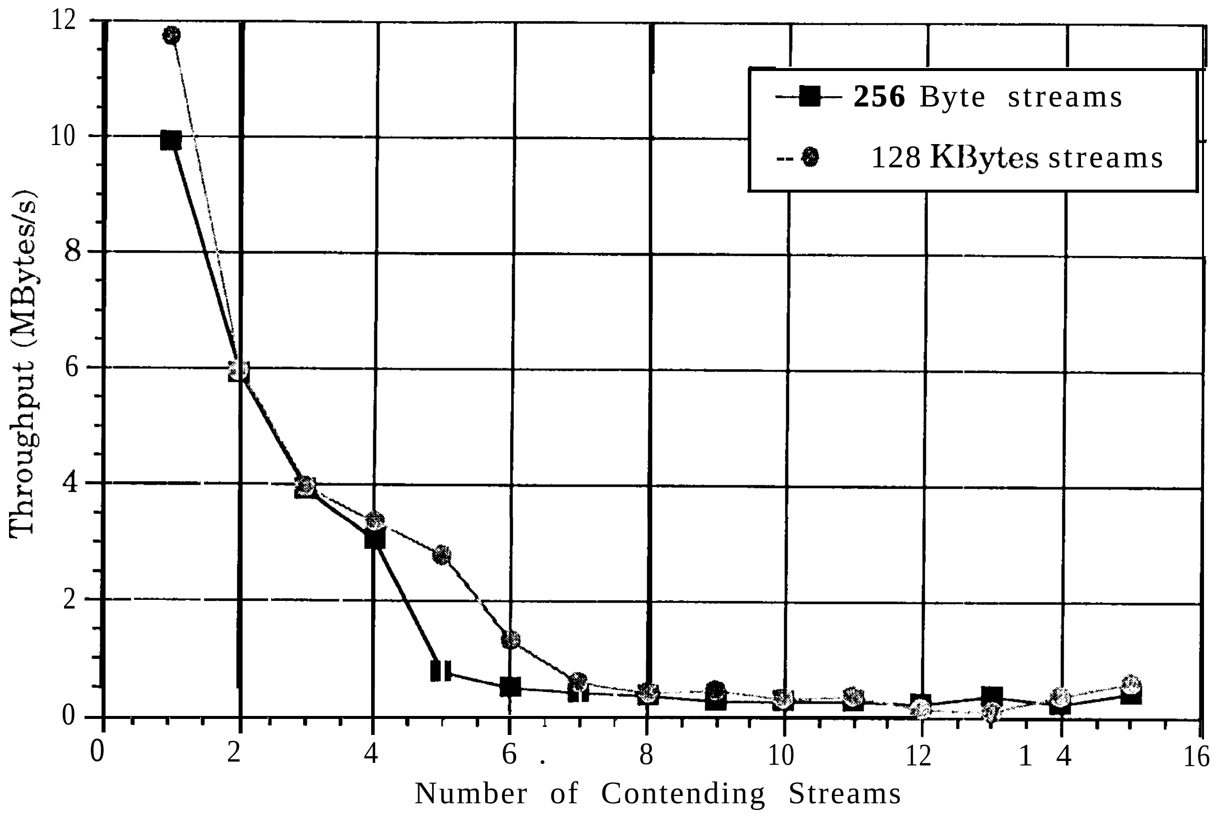


Figure 2. Traffic loading effects on throughput performance

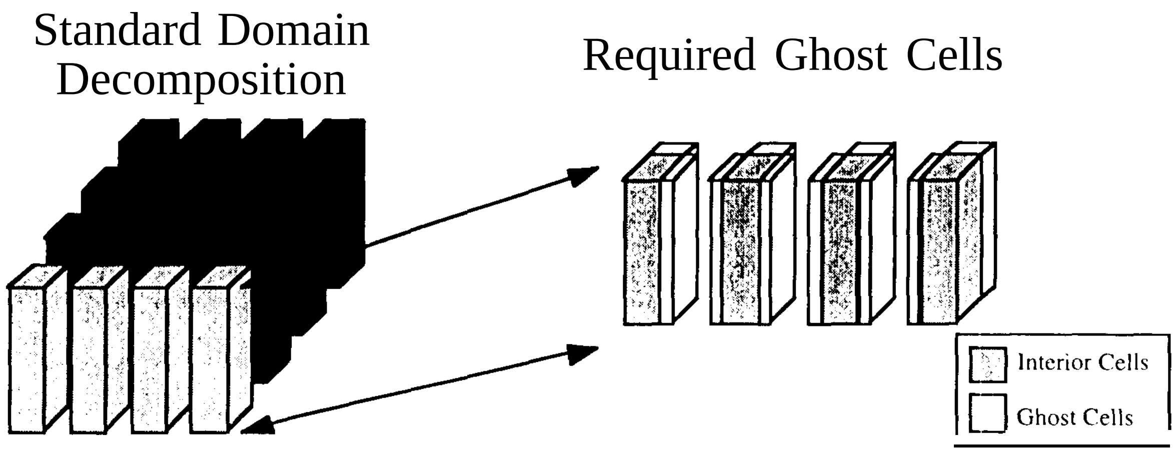


Figure 3. The relation between the 2-D decomposition of the 3-D grid and the required ghost cell communication.

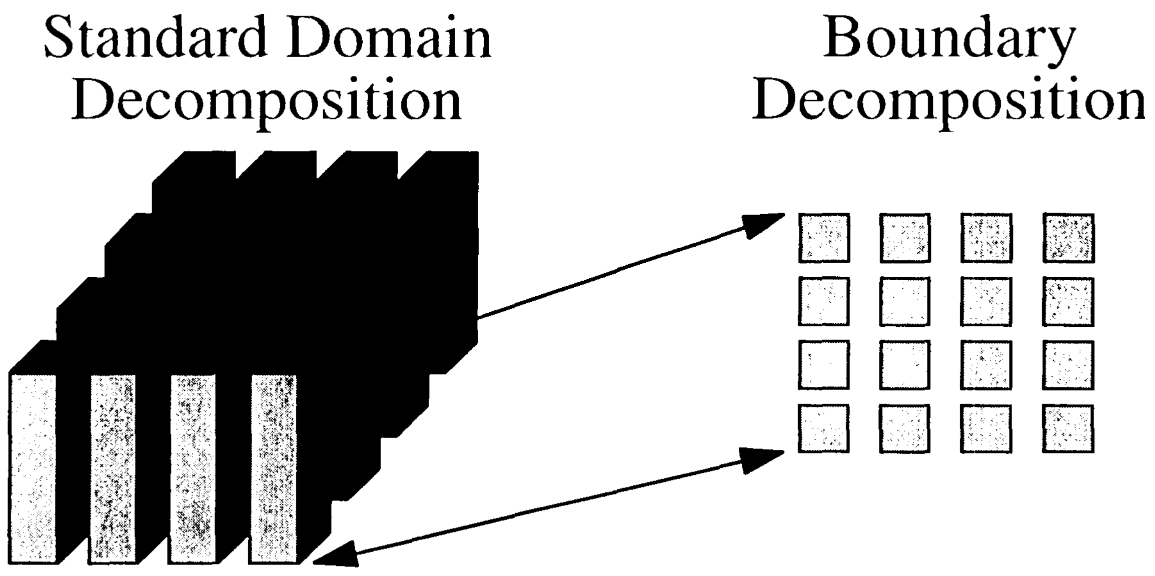


Figure4. Therelation between the2-Ddecomposition of the3-Dgrid and the possible redistribution of the boundary variables.

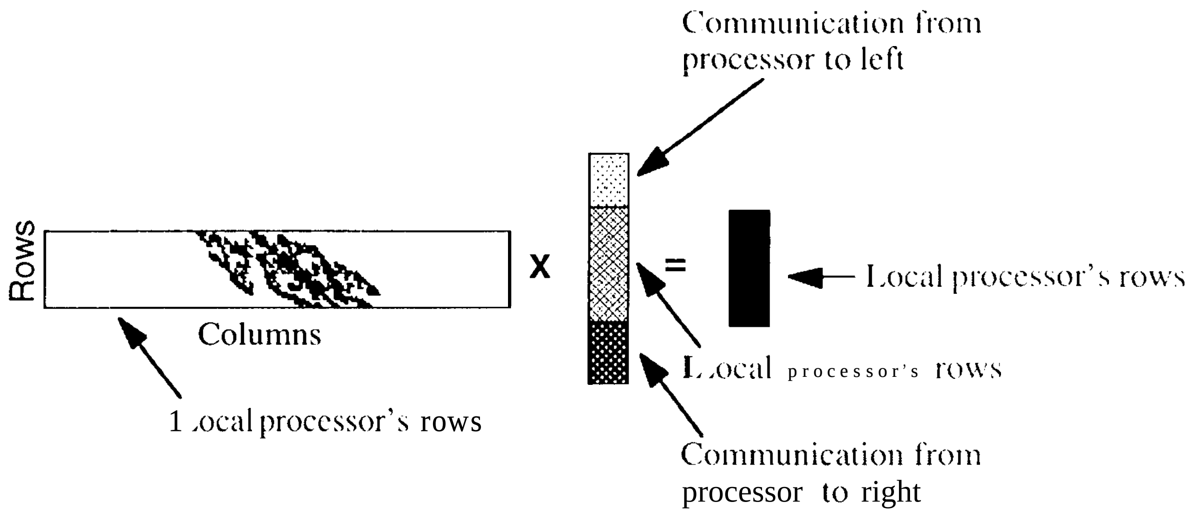


Figure 5. A representation of the work required in each processor to perform a matrix-vector multiply.

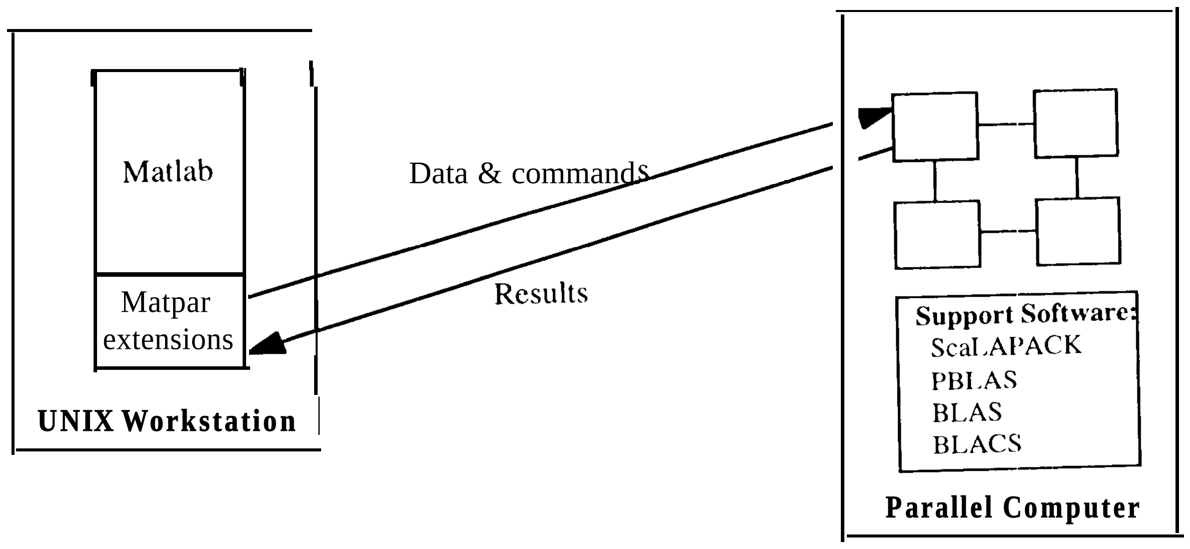


Figure 6. Matpar Architecture